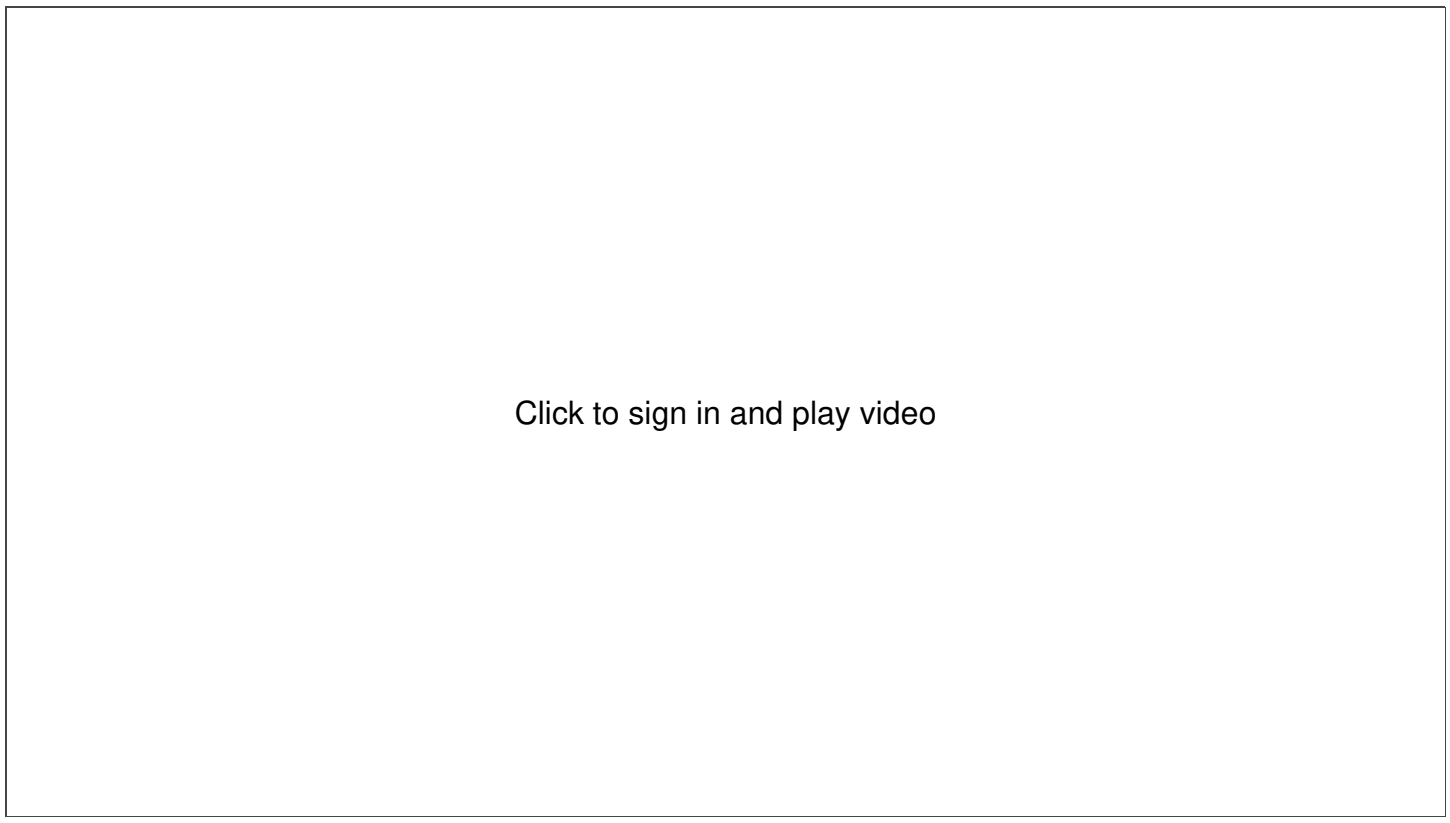
Dynamic Memory Management

MONDAY, MAY 5

An introduction to the world of *dynamic memory*, where we as programmers now have complete control over how computer memory is used. We create for the first time objects that can live beyond the lifespans of the functions where they are created.

- 📚 Readings: Text 12.1, 12.3, 14.2
- 📝 Lecture quiz on Canvas
- 🎬 Lecture video on Canvas

Lecture Video

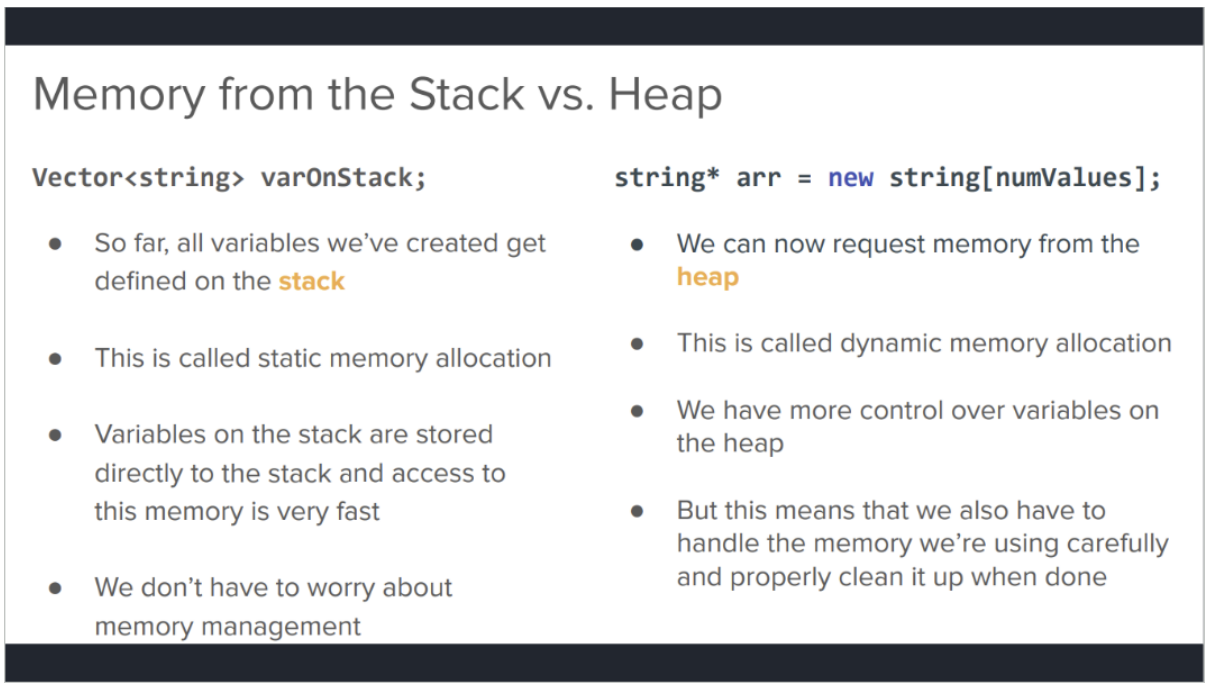


Contents

Overview

We talked today about *dynamic memory allocation*, which allows us to set aside memory where we can store data that lives beyond the lifespan of a function. The trade-off here is that we must keep careful tabs on dynamically allocated memory and manually release it when we're finished using it order to prevent *memory leaks* -- situations where our programs hog memory they're not using really using anymore, which can potentially place a serious strain on system resources.

The following slide gives a summary of key differences between statically allocated memory (stack space) and dynamically allocated memory (heap space):



Credit: Kris Jeong (Section Leader, Fall '23)

Motivation

We started today with a familiar example from last Friday's lecture:

```
Vector<int> createRandoVector(int n)
{
   Vector<int> v;

   for (int i = 0; i < n; i++)
   {
      v.add(randomInteger(1, 100));
   }

   return v;
}
```

Recall that the locally declared `v` vector dies when we leave this function. What is returned to whoever calls this function is a *copy* of `v`. That leads to a few problems:

1. Since returning the vector by value creates a copy of that vector, returning is a slow operation. We have to copy every single value from that vector into a new copy when we return from this function.
2. If we try to return a reference or pointer, we will be returning a reference or pointer to a dead variable. Using that reference or pointer to access the vector from outside this function would crash our program.

Today, we ultimately solved this problem by doing the following:

1. Using `new` to dynamically allocate a space in memory that lives beyond the lifespan of the function where it's set aside.
2. Returning a pointer to that dynamically allocated memory space, which is super fast because pointers are typically represented using only 64 bits on most systems. That's not much data to return from a function at all.

Proof That Local Variables Die When We Leave a Function (Part 1 of 2)

Using our new knowledge of pointers, we were able to prove that returning a vector by value creates a new copy of that vector. Check this out:

```
#include <iostream>
#include "console.h"
#include "random.h"
#include "vector.h"
using namespace std;

Vector<int> createRandoVector(int n)
{
   Vector<int> v;

   for (int i = 0; i < n; i++)
   {
      v.add(randomInteger(1, 100));
   }

   cout << "Address of vector in createRandoVector(): " << &v << endl;
   cout << "Contents of vector in createRandoVector(): " << v << endl;

   // This returns a copy of our vector. This is a slow, expensive operation.
   return v;
}

int main()
{
    Vector<int> v;

    cout << "Address of vector in main() before calling cRV(): " << &v << endl;

    v = createRandoVector(5);

    // Notice the address of our local v variable does not change.
    cout << "Address of vector in main() after calling cRV(): " << &v << endl;
    cout << "Contents of vector in main(): " << v << endl;

    return 0;
}
```

memory diagram -- immediately before returning from createRandoVector():

```
createRandoVector():

  x        0x7f3118dbbc00  <-- different address from the one in main()!
  +---------------------+
  |  {9, 27, 3, 1, 91}  |
  +---------------------+

main():

  x        0x7f3118dbbbd0
  +---------------------+
  |  {9, 27, 3, 1, 91}  |
  +---------------------+


(*) This memory diagram is somewhat misleading. The array within the vector actually
    exists in heap space. :-o That detail isn't particularly relevant or important to
    us at this moment, though.
```

output:

```
Address of vector in main() before calling cRV(): 0x7f3118dbbbd0
Address of vector in createRandoVector(): 0x7f3118dbbc00
Contents of vector in createRandoVector(): {9, 27, 3, 1, 91}
Address of vector in main() after calling cRV(): 0x7f3118dbbbd0
Contents of vector in main(): {9, 27, 3, 1, 91}
```

The output above shows that the vector in `createRandoVector()` is stored at a different address from the one in `main()`. This provides evidence that what is returned from the `createRandoVector()` function is just a *copy* of the vector it created (and remember, copy = slow).

We're close to being able to solve this problem!

Quokka Destruction: Proof That Local Variables Die When We Leave a Function (Part 2 of 2)

Today, I used output from constructor and destructor functions as an alternative way to show that the objects created within a function die when we leave that function. I injected a bunch of `getLine()` function calls into the program so we could walk through the program step by step and see what was happening where.

quokka.h:

```
#ifndef QUOKKA_H
#define QUOKKA_H

#include <iostream>

class Quokka
{
public:
   Quokka();
   Quokka(std::string name);
   ~Quokka();

private:
   std::string _name;
};

#endif  // QUOKKA_H
```

quokka.cpp:

```
#include <iostream>
#include "quokka.h"
using namespace std;

Quokka::Quokka()
{
}

Quokka::Quokka(string name)
{
   _name = name;
   cout << "Hello, " << _name << endl;
}

Quokka::~Quokka()
{
   cout << "R.I.P. " << _name << endl;
}
```

main.cpp:

```
#include <iostream>
#include "console.h"
#include "quokka.h"
#include "simpio.h"
using namespace std;

void createQuokka()
{
   cout << "In createQuokka()..." << endl;
   getLine();

   Quokka q("Muffinface");

   cout << "About to leave createQuokka()..." << endl;
   getLine();
}

int main()
{
   cout << "About to call createQuokka()..." << endl;
   getLine();

   createQuokka();

   cout << "Back in main()..." << endl;
   getLine();

   return 0;
}
```

output:

```
About to call createQuokka()...

In createQuokka()...

Hello, Muffinface
About to leave createQuokka()...

R.I.P. Muffinface
Back in main()...
```

Notice above that as soon as we leave `createQuokka()`, before getting back to `main()` and printing `Back in main()...`, our quokka is deconstructed. Again, this provides evidence that local variables die when we leave a function.

The Immortal Quokka (and C++'s `new` Operator) (and Dynamic Memory Allocation)

Our next goal was to create an immortal quokka -- one that wouldn't die when we left the function where it was created.

For that, we used the `new` operator, which sets aside memory for whatever data type we would like to hold and returns the address of that memory block. Aha! Now you know why we talked about pointers on Friday! We need a place to store the memory address returned by `new`, which means we need pointers!

The syntax for `new` is as follows:

```
new DATA_TYPE;
```

That will return a pointer to the newly allocated memory space, so we typically capture the result in a pointer variable:

```
DATA_TYPE *var = new DATA_TYPE;
```

If we want to create an array in this fashion, the syntax is:

```
DATA_TYPE *var = new DATA_TYPE[ARRAY_LENGTH];
```

Here are some key points about `new`:

- (*Key take-away!*) The memory we set aside with `new` exists outside the stack frame for our function call, which means it will not die when we leave our function. Cool! Instead of existing in *stack space*, the stuff we create with `new` exists in what we call *heap space*.
- (*Key take-away!*) Because the memory set aside with `new` is not released automatically when we leave the function where that happened, it's up to us to keep track of that memory and manually release it to the system using the `delete` operator when we're finished with it. Failure to do so will lead to a *memory leak* (discussed more in a section of notes below).
- (*Key take-away!*) Remember that `new` returns to us the address of the block of memory it just set aside for us in heap space. We store that address in a pointer variable.
- (*Key take-away!*) The memory we set aside with `new` has behaviors that are not predicted at compile-time. If we use `new` to create an array, the length of that array might be based on some variable whose value is determined through user input or file input at runtime. Furthermore, the exact place where we finish using that memory and manually release it back to the system might be governed by user input that cannot be predicted at compile-time. The behaviors here are *dynamic*.
- (*Key take-away!*) Accordingly, when we set aside memory with `new`, we call that **dynamic memory allocation**. (In contrast, the creation of local variables without the use of `new` is called **static memory allocation**, which is a bit obnoxious because the word "static" has additional meanings in C++ and many other languages.)

Here's how we modified our `createQuokka()` function to create a quokka that lived beyond the lifespan of our function. Notice the following changes:

1. We use `new` to create our quokka.
2. We store our result in a pointer variable: `Quokka *`.
3. If we want to ship that pointer back to `main()`, we have our function return a `Quokka *` as well.

main.cpp:

```
#include <iostream>
#include "console.h"
#include "quokka.h"
#include "simpio.h"
using namespace std;

void createQuokka()
{
   cout << "In createQuokka()..." << endl;
   getLine();

   Quokka *q = new Quokka("Muffinface");

   cout << "About to leave createQuokka()..." << endl;
   getLine();
}

int main()
{
   cout << "About to call createQuokka()..." << endl;
   getLine();

   createQuokka();

   cout << "Back in main()..." << endl;
   getLine();

   return 0;
}
```

output:

```
About to call createQuokka()...

In createQuokka()...

Hello, Muffinface
About to leave createQuokka()...

Back in main()...
```

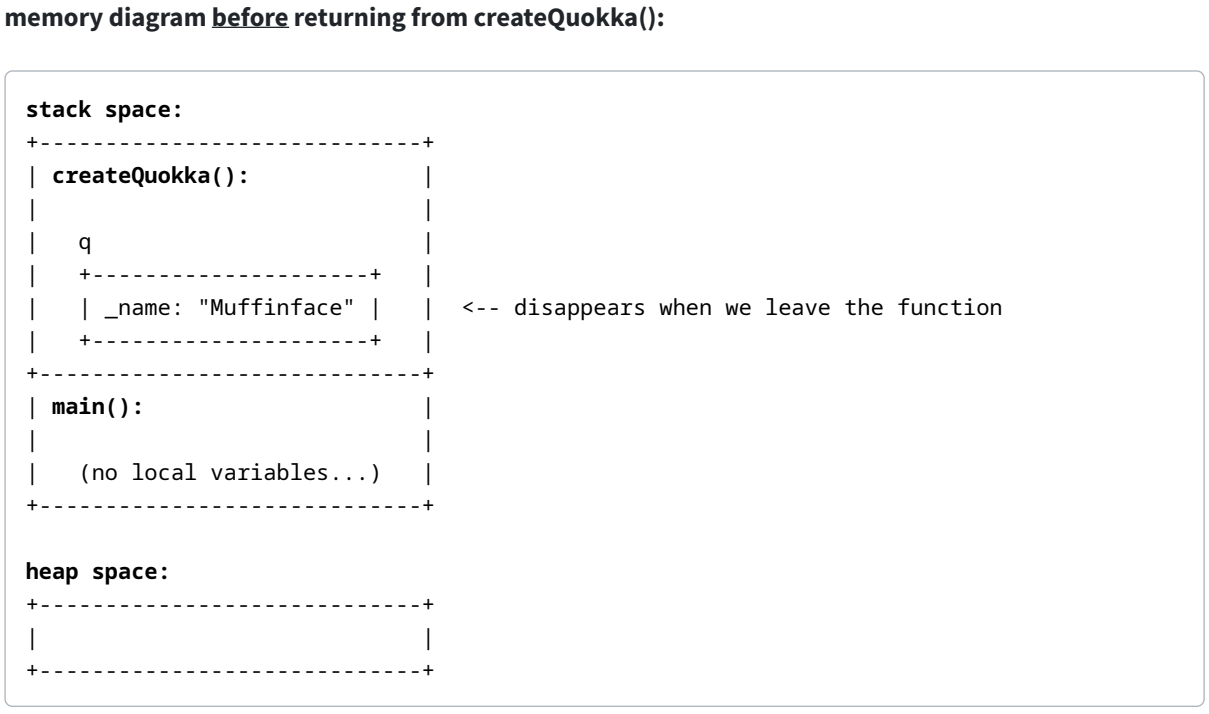
Notice that we no longer see the `R.I.P. Muffinface` line in the program's output! That means our quokka destructor is not getting called when we leave `createQuokka()`!

✨ **Mission accomplished!** ✨ We've created an object that lives beyond the lifespan of the function where it was created. Neat!

Memory Diagrams: Static vs. Dynamic Quokka Allocation

Our original `createQuokka()` function was creating our new quokka in stack space. So, when we left the function, our quokka died:

memory diagram before returning from createQuokka():



memory diagram after returning from createQuokka():

```
stack space:
+-----------------------------+
| main():                     |
|                             |
| (no local variables...)     |
+-----------------------------+

heap space:
+-----------------------------+
|                             |
+-----------------------------+
```

Compare this to the `createQuokka()` function that uses dynamic memory allocation with `new`. In that case, our quokka exists even after we leave the function:

memory diagram before returning from createQuokka():

```
stack space:
+-----------------------------+
| createQuokka():             |
|                             |
|   q                         |
|   +---------------------+   |
|   |     0x019247c00     |   |  <-- memory address of dynamically allocated quokka
|   +---------------------+   |
+-----------------------------+
| main():                     |
|                             |
|   (no local variables...)   |
+-----------------------------+

heap space:
+-----------------------------+
|   0x019247c00               |
|   +---------------------+   |
|   | _name: "Muffinface" |   | <-- lives even when we leave the function
|   +---------------------+   |
+-----------------------------+
```

memory diagram after returning from createQuokka():

```
stack space:
+-----------------------------+
| main():                     |
|                             |
|   (no local variables...)   |
+-----------------------------+

heap space:
+-----------------------------+
|   0x019247c00               |
|   +---------------------+   |
|   | _name: "Muffinface" |   | <-- still here! (but inaccessible from main()!)
|   +---------------------+   |
+-----------------------------+
```

Concerns About the Immortal Quokka (and Memory Leaks!)

At this point, you might be tremendously concerned that the immortal quokka we created in `createQuokka()` will take up memory in our system forever and ever. That's not really the case. When a program terminates -- when we finally return from `main()` -- all memory associated with that program is reclaimed by the operating system. So, while our quokka lived beyond the lifespan of the function where it was created, it didn't live beyond the lifespan of our whole *program*. (If that were possible, the consequences for memory management would be disastrous.)

It's considered poor form, however, to let a program terminate without having *manually* released our claim to all the dynamically allocated memory we set aside over the course of our program. We should do our own bookkeeping and release dynamically allocated memory once we no longer need it. When we fail to do so -- or when we completely lose all record of a pointer given to us by `new` and can therefore no longer access that memory address -- we have what we call a **memory leak** or **orphaned memory**.

Memory leaks are especially problematic if a program is expected to run for a very long time without being restarted. If it has a tiny memory leak where it dynamically allocates memory and either (a) does not release that memory block when it's finished using it, or (b) loses the pointer to that memory block altogether before releasing it, then the program could slowly sap our system resources over time by filling up memory with useless junk,

causing our entire system to slowly grind to a halt.

For example:

main.cpp:

```
#include <iostream>
#include "console.h"
#include "quokka.h"
#include "simpio.h"
using namespace std;

void createQuokka()
{
   cout << "In createQuokka()..." << endl;
   getLine();

   // Dynamic memory allocation!
   Quokka *q = new Quokka("Muffinface");

   cout << "About to leave createQuokka()..." << endl;
   getLine();
}

int main()
{
   cout << "About to call createQuokka()..." << endl;
   getLine();

   // YIKES! This creates a memory leak! A call to this function creates a Quokka
   // object in heap space, but we have no way to get to that object because we
   // failed to return its address from createQuokka(). That object will continue
   // to take up memory -- even though we can no longer reach it -- until this
   // program terminates.
   createQuokka();

   cout << "Back in main()..." << endl;
   getLine();

   return 0;
}
```

We can create an even nastier memory leak if we do something like this:

```
int main()
{
   // Create ONE BILLION Quokka objects!
   for (int i = 0; i < 1000000000; i++)
   {
      // The first iteration of this loop creates a memory leak! See diagrams below
      // for explanation.
      createQuokka();
   }

   return 0;
}
```

Here's what's happening in memory as we perform our first few iterations of that loop:

memory after our first call to createQuokka():

```
stack space:
+----------------------------+
| main():                    |
|                            |
|   (no local variables...)  |
+----------------------------+

heap space:
+----------------------------+
|   0x019247c00              |
|   +--------------------+   |
|   | _name: "Muffinface" |   | <-- createQuokka() just set aside this space. We cannot
|   +--------------------+   |     reach it from main() because we did not return this
+----------------------------+     pointer from createQuokka(). So, we already have a
                                   memory leak!
```

memory after our second call to createQuokka():

```
stack space:
+-----------------------------+
| main():                     |
|                             |
|   (no local variables...)   |
+-----------------------------+

heap space:
+-----------------------------+
|   0x019247c00               |
|   +---------------------+   |
|   | _name: "Muffinface" |   |
|   +---------------------+   |
|                             |
|   0x01924d440               |
|   +---------------------+   |
|   | _name: "Muffinface" |   |  <-- We now have a SECOND quokka lingering in memory!
|   +---------------------+   |
+-----------------------------+
```

memory after our third call to createQuokka():

```
stack space:
+-----------------------------+
| main():                     |
|                             |
|   (no local variables...)   |
+-----------------------------+

heap space:
+-----------------------------+
|   0x019247c00               |
|   +---------------------+   |
|   | _name: "Muffinface" |   |
|   +---------------------+   |
|                             |
|   0x01924d440               |
|   +---------------------+   |
|   | _name: "Muffinface" |   |
|   +---------------------+   |
|                             |
|   0x01924fe00               |
|   +---------------------+   |
|   | _name: "Muffinface" |   |  <-- We now have a THIRD quokka lingering in memory!
|   +---------------------+   |      This is getting out of control!
+-----------------------------+
```

That's not good. Let's explore how to squash that memory leak.

Returning Dynamically Allocated Variables from Functions

To squash the memory leak created by our repeated calls to `createQuokka()`, we need that function to return a pointer to the dynamically allocated memory it has set aside. For that, we just return `q`. Recall that our variable `q` contains the memory address we got from `new`. By returning `q`, we are returning that memory address to `main()`. To facilitate that return, we need to update the return type of `createQuokka()` to match the type of the variable being returned. In this case, that's a `Quokka *`.

```
Quokka *createQuokka()
{
   cout << "In createQuokka()..." << endl;
   getLine();

   Quokka *q = new Quokka("Muffinface");

   cout << "About to leave createQuokka()..." << endl;
   getLine();

   return q;
}
```

Note that this is a super fast `return` statement! We are no longer returning a *copy* of the thing we created in our function. We are instead returning a tiny, 64-bit memory address. This is enormously fast and can be used to fully resolve the issues we had with our `createRandoVector()` function at the top of today's notes.

We then need to capture those pointers in a variable in `main()` and free them up before moving on to the next iteration of our loop. The syntax for releasing dynamically allocated memory back to our system (often called "freeing" that memory), is as follows:

```
delete  POINTER_TO_DYNAMICALLY_ALLOCATED_SPACE ;
```

For dynamically allocated arrays, we add `[]` brackets to the end of delete, like so:

```
delete[]  POINTER_TO_DYNAMICALLY_ALLOCATED_ARRAY ;
```

Recall from class that `new` and `delete` are in cahoots. `new` gives us an address for our dynamically allocated block of memory. We call `delete` on such an address to tell our memory management unit that we no longer lay any claim over that block of memory.

Here is our updated `main()`, which no longer has a memory leak:

```
int main()
{
   // Create ONE BILLION Quokka objects!
   for (int i = 0; i < 1000000000; i++)
   {
      // We now have a box called q that holds the address of our dynamically
      // allocated quokka!
      Quokka *q = createQuokka();

      // This releases the memory associated with the address stored in q!
      // Goodbye, memory leak! (This also causes our Quokka object's destructor
      // function to be called!)
      delete q;
   }

   return 0;
}
```

Super Important Rule of Thumb: For Every `new`, a Single `delete`

(*Important note!*) A super important rule of thumb is that for every `new` statement in a program, we should have a corresponding `delete` statement to free up the dynamically allocated memory, and that should happen *before* we lose access to or overwrite the last remaining pointer we have to that space in memory.

Super Important Note: Do Not Attempt to Dereference an Address After You `delete` It! 😳

(*Important note!*) Recall that `delete ptr` does not delete the `ptr` variable. It simply releases any claim we have over the memory that `ptr` is pointing to, signaling to our OS that the memory chunk in question can now be used for other things.

After we `delete ptr`, the `ptr` variable will still have the address of that dynamically allocated chunk of memory we just released. Attempting to dereference `ptr` at that point will take us back to that address, but could get us into all kinds of wacky trouble! There's certainly a chance that if we returned to that memory address, we would find the data we had just left there; the OS doesn't necessarily overwrite it with zeros or otherwise clear it out. But we could also find that the memory is already being used for something else. We have absolutely no claim over that chunk of memory, and so it is super dangerous to return there.

Returning to a memory address after we `delete` it is kind of like going back to a trash heap, retrieving a coffee cup that has been sitting there for a few hours, and taking a sip. Sure, there might still be some coffee in there, but it might also be spoiled or contaminated at that point, and that's also just super gross. **DON'T DO IT!!** 🤢😳🤮

Array-Based Stack

We then combined the topics from this lecture, our lecture on pointers, and our lecture on object-oriented programming in order to implement an array-based stack that expands automatically to accommodate new elements.

Some key aspects of this implement are as follows:

1. We use `new` and `delete` to create and destroy arrays every time the stack needs to expand.
2. The syntax for freeing an entire array is `delete[] myArray;`. Note the need for the `[]` brackets in that statement.
3. Recall that we use an `int *` to point to the base of an array. To some degree, an `int *` can be treated as an integer array. We can we apply an offset such as `[i]` to an `int *` variable, and that pointer will be dereferenced by those brackets in addition to our system skipping forward `i` cells in memory.

The stack implementation we created is as follows:

arraybasedstack.h:

```
#ifndef ARRAYBASEDSTACK_H
#define ARRAYBASEDSTACK_H

// Starting with a ridiculously small initial capacity so we can observe the
// expansion of the stack in a small test case.
#define DEFAULT_STACK_CAPACITY 3

class ArrayBasedStack
{
public:
   ArrayBasedStack();
   ~ArrayBasedStack();
   void push(int value);
   int pop();
   int peek() const;
   int size() const;
   bool isEmpty() const;

private:
   int *_elements;
   int _size;
   int _capacity;
};

#endif // ARRAYBASEDSTACK_H
```

arraybasedstack.cpp:

```
#include <iostream>
#include "arraybasedstack.h"
#include "error.h"
using namespace std;

ArrayBasedStack::ArrayBasedStack()
{
   // Creating this dynamically ensures it lives when we leave our constructor
   // function, which is definitely the desired behavior here!
   _elements = new int[DEFAULT_STACK_CAPACITY];
   _size = 0;
   _capacity = DEFAULT_STACK_CAPACITY;

   cout << "Created stack with capacity: " << _capacity << endl;
}

ArrayBasedStack::~ArrayBasedStack()
{
   // When it's time to destroy a stack, we must free the array it contains.
   // Otherwise, we will lose access to that array's address and have a memory
   // leaks that persists until our program terminates. Recall that to delete
   // an array, we need [] brackets after "delete".
   delete[] _elements;
}

void ArrayBasedStack::push(int value)
{
   // If our stack is full, we must expand the underlying array before expansion.
   // It's probably better design to outsource this to a private member function,
   // but I didn't take the time to do that in class.
   if (_size >= _capacity)
   {
      // Create new, larger stack.
      int *newArray = new int[_capacity * 2 + 1];

      // Copy elements from old array into new array.
      for (int i = 0; i < _size; i++)
      {
         newArray[i] = _elements[i];
      }

      // Free old array. If we don't do this before setting _elements = newArray,
      // we will lose this pointer forever and have a memory leak!
      delete[] _elements;

      // Set the stack's internal pointer to point to the new array.
      _elements = newArray;

      // Update the stack capacity.
      _capacity = _capacity * 2 + 1;

      cout << "Expanded stack to capacity: " << _capacity << endl;
   }

   _elements[_size] = value;
   _size++;
}

int ArrayBasedStack::pop()
{
   if (isEmpty())
   {
      error("Empty stack in pop()!");
   }

   int result = _elements[_size - 1];
   _size--;
   return result;
}

int ArrayBasedStack::peek() const
{
   if (isEmpty())
   {
      error("Stack is empty in peek()!");
   }

   return _elements[_size - 1];
}

int ArrayBasedStack::size() const
{
   return _size;
}

bool ArrayBasedStack::isEmpty() const
{
   return _size == 0;
}
```

main.cpp:

```
#include <iostream>
#include "arraybasedstack.h"
#include "console.h"
using namespace std;

int main()
{
   ArrayBasedStack s;

   for (int i = 0; i < 10; i++)
   {
      s.push(i);
   }

   while (!s.isEmpty())
   {
      cout << s.pop() << endl;
   }

   return 0;
}
```

output:

```
Created stack with capacity: 3
Expanded stack to capacity: 7
Expanded stack to capacity: 15
9
8
7
6
5
4
3
2
1
0
```

`const` Member Functions

You're also seeing this in section this week, so I don't feel too bad about mentioning this only briefly in class today, but: when we have a member function that isn't supposed to change the internal state of an object at all (such as our `peek()` function above, which should return a value without modifying the stack), it's conventional to put `const` after the function name in both our header file and our .cpp file. That actually ensures that the function cannot change any of our member variables (if you try, the code won't compile), and this is considered a best practice so that no one can come along and accidentally modify the code to change member variables that shouldn't have been changed.

What's next?

On Wednesday, we will shift gears a bit and look at a new data structure: the minheap, which is used to implement priority queues. After that, we will talk briefly about sorting algorithms. Next week, we will delve into nested and linked dynamically allocated structures, and we will explore a data structure that relies heavily on pointers and dynamic memory management: linked lists.

Exercises

1. Trace through what's happening with the dynamic memory management in today's `ArrayBasedStack` class for the small example given in that program's `main()` function. Draw memory diagrams that distinguish between variables in stack space and heap space.

2. After reviewing today's notes, implement an array-based stack from scratch. Then do the same with an array-based queue and an array-based vector. Inject `cout` statements above each `new` and `delete` statement so you can see when you run your program that each invocation of the `new` operator has a corresponding invocation of the `delete` operator.

3. With today's array-based stack, what reason did I give for using the formula `newSize = oldSize * 2 + 1` instead of `newSize = oldSize * 2` when expanding the stack?

4. When creating our array-based stack, why might we not want to just always start with an arbitrarily huge array so that we don't have to worry about dynamic expansion operations?

5. As always, the textbook and this week's section are chock full of great exercises and additional examples to help reinforce this material. In particular, be sure to check out the NotoriousRBQ (ring buffer queue) in this week's section.

Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-May-05